Application Note

BCM43XX

Wireless Device Driver Porting

43XX-AN1300-R

5300 California Avenue • Irvine, CA 92617 • Phone: 949-926-5000 • Fax: 949-926-5203 May 13, 2011

Revision History

Revision	*Date*	*Change Description*
43XX-SDM1300-R	05/13/11	**Updated:** • "Terminology" on page 6 • "GMODE" on page 15 • "LEDs" on page 17
43XX-SDM102-R	03/26/04	Minor edits.
43XX-SDM101-R	02/04/04	Several minor updates.
43XX-SDM100-R	10/17/02	Initial release.

Broadcom Corporation
5300 California Avenue
Irvine, CA 92617

Table of Contents

List of Tables

About This Document

Purpose and Audience

This document explains the process of porting the Broadcom wireless driver to customer-specific platform, operating system (OS), and CPU.

This document is intended specifically for customers who have the full source access to the Broadcom driver (under SLA) and are using the Broadcom client chipsets to integrate into their own CPU and OS environment.

Acronyms and Abbreviations

In most cases, acronyms and abbreviations are defined on first use.

For a comprehensive list of acronyms and other terms used in Broadcom documents, go to: http://www.broadcom.com/press/glossary.php.

Document Conventions

The following conventions may be used in this document:

Convention	*Description*
Bold	User input and actions: for example, type **exit**, click **OK,** press **Alt+C**
`Monospace`	Code: `#include <iostream>` HTML: `<td rowspan = 3>` Command line commands and parameters: `wl [–l] <command>`
< >	Placeholders for *required* elements: enter your <username> or `wl <command>`
[ ]	Indicates *optional* command-line parameters: `wl [–l]` Indicates bit and byte ranges (inclusive): [0:3] or [7:0]

Technical Support

Broadcom provides customer access to a wide range of information, including technical documentation, schematic diagrams, product bill of materials, PCB layout information, and software updates through its customer support portal (https://support.broadcom.com). For a CSP account, contact your Sales or Engineering support representative.

In addition, Broadcom provides other product support through its Downloads & Support site (http://www.broadcom.com/support/).

We welcome your feedback on improving the portability of the wireless device driver and the usability of this document. Send your comments to hnsupport@broadcom.com.

Terminology

In IEEE 802.11, the addressable unit is a station (STA). The Basic Service Set (BSS) is the building block of an IEEE 802.11 LAN, similar to a coverage area. Each BSS consists of one or more STAs. The Independent Basic Service Set (IBSS) is the most basic type of IEEE 802.11 LAN. This mode occurs when stations communicate directly with one another and is often referred to as an ad hoc network. To become a member of a BSS infrastructure, the STA must associate with the BSS. Multiple BSSs may be interconnected to create a Distribution System (DS). An Access Point (AP) is a STA that also provides access to the DS.

The IEEE 802.11b specification is based on the Complementary Code Keying (CCK) physical layer protocol in the 2.4 GHz frequency band (channels 1 through 14) and supports {1, 2, 5.5, 11} Mbit/s rate encodings.

The IEEE 802.11g specification resides in the same 2.4 GHz frequency band, but is based on Orthogonal Frequency Division Multiplexing (OFDM) physical layer and supports {1, 2, 5.5, 6, 9, 11, 12, 18, 24, 26, 48, and 54} Mbit/s data rates and backward compatible with IEEE 802.11b on 1, 2, 5.6, and 11 Mbps rates.

The IEEE 802.11a specification is based on the OFDM physical layer in the 5.0 GHz frequency band (channels 36, 40, 44, 48, 52, 60, 64, 149, 153, 157, 161) and supports {6, 9, 12, 18, 24, 36, 48, 54} Mbps rate encodings.

The IEEE 802.11n specification is an amendment to the IEEE 802.11 wireless networking standard to improve network throughput over the two previous standards—IEEE 802.11a and IEEE 802.11g—with a significant increase in the maximum raw data rate from 54 Mbit/s to 600 Mbit/s with the use of four spatial streams at a channel width of 40 MHz.

All BCM43XX semiconductors includes dot11 a/b/g/n core, PCI/PCIE cores.

A coreidx (core index) is the index, starting at zero, of a core within the Silicon Backplane enumeration space. It is a positive integer in the range 0 through SB_MAXCORES.

A coreunit is the index, starting at zero, of a core among one or more cores of the same type. For example, if a chip contained two d11 cores, the first d11 would be coreunit 0 and the second would be coreunit 1. The coreunit and the coreidx are orthogonal concepts.

The A band is a set of radio channels. The B band is centered at 2.4 GHz. The A band is centered at 5 GHz. Board vendors, Boardtypes, PCI device IDs, and so on are numeric values representing particular companies, board designs and different chip types with values #defined by bcmdevs.h.

Boardstyle is a numeric value representing the board bus type with values #defined by bcmdefs.h:

```
#define SI_BUS        0      /* SOC Interconnect */
#define PCI_BUS       1      /* PCI target */
#define PCMCIA_BUS    2      /* PCMCIA target */
#define SDIO_BUS      3      /* SDIO target */
```

Source Directories

Table 1: Source Directories

File	*Description*
src/include	Driver-independent header files
src/include/proto	Over-the-wire protocol definitions
src/shared	Driver-independent shared source files
src/wl/sys	Wireless driver source directory
src/wl/phy	Wireless driver phy related source files
src/wl/exe	Wireless command tool source
src/include/bcmcrypto	Security-related definitions
src/bcmcrypto	Security-related source files

Source Files

File	*Description*
src/include/typedefs.h	A small set of type definitions used by all Broadcom Home Networking Division (HND) software
src/include/bcmendian.h	Defines some little-/big-endian macros
src/include/d11.h	BCM43XX dot11 core hardware register declarations, defines, and so on.
src/include/osl.h	OS abstraction layer main header wrapper
src/include/epivers.h	Driver version information
src/include/xx_osl.h	OS specific OSL definitions
src/include/hnddma.h	HND DMA hardware/software module definitions
src/include/bcmutils.h	Miscellaneous useful driver-independent macros and definitions
src/include/siutils.h	Silicon Backplane software utility module definitions
src/include/sbconfig.h	Core-independent Silicon Backplane definitions
src/include/hndpci.h	SB PCI core hardware definitions
src/include/sbpcmcia.h	SB PCMCIA core hardware definitions
src/include/bcmsdh.h	SDIO bus access routines (stubbed out)
src/include/wlioctl.h	Wireless driver I/O control API
src/include/bcmdevs.h	Known chip PCI vendor and device ID definitions
src/include/bcmnvram.h*	NVRAM variables manipulation definitions
src/include/bcmsrom.h*	NIC SROM routine definitions
src/include/bcmcrypto/	Contains header files for c files in src/bcmcrypto/

File	*Description*
src/include/proto/802.11.h	IEEE 802.11 protocol specification definitions

File	*Description*
src/shared/hnddma.c	Broadcom HND DMA code
src/shared/xx_osl.c	OS-specific OSL implementations
src/shared/sbutils.c	Silicon Backplane software utility code
src/shared/nvramstubs.c	NVRAM shared code
src/shared/bcmsrom.c	SROM shared code
src/shared/bcmutils.c*	General OS-independent utilities
src/shared/flashutl.c*	Flash read, write, and erase routines
src/shared/nvramstubs.c*	Stubs for flash-based nvram_get function
src/shared/hndmips.c*	Silicon Backplane MIPS core routines
src/shared/hndpci.c*	Low-level PCI and silicon backplane support for BCM47xx
src/shared/sflash.c*	Silicon backplane chip common serial flash interface

File	*Description*
src/wl/sys/wl_xx.c	OS-specific wireless driver ports
src/wl/sys/wl_xx.h	OS-specific wireless driver tunable definitions
src/wl/sys/wlc.[ch]	Primary common (OS-independent) wireless driver source file
src/wl/phy/*	Common wireless driver PHY/radio-specific routines
src/wl/sys/wlc_ap.[ch]	Small portion of wireless common AP-specific code, mostly PS features
src/wl/sys/d11ucode.[ch]	Precompiled MAC microcode
src/wl/sys/wl_export.h	Prototypes of functions each wl_xx.c must export
src/wl/sys/wl_dbg.h	#includes the port-specific wl_xx.h file and defines some wireless-specific debug macros
src/wl/sys/wlc_security.[ch]	Advanced wireless security protocols (under development)
src/wl/exe/wlu.c	Primary common (OS-independent) wireless command tool file
src/wl/exe/wlu_xx.c	OS-specific support functions for wireless command tool
src/wl/sys/wlc_ethereal.h	Ethereal definitions
src/wl/sys/wlc_led.[ch]	LED management and maintenance
src/wl/sys/wlc_rate.[ch]	Common code for rate-selection algorithms
src/wl/bcmcrypto/tkhash.c*	Generate 802.11 per-packet RC4 key hash test vectors
src/bcmcrypto/aes.c*	AES encryption decryption wrapper functions
src/bcmcrypto/tkmic.c	TKIP Message Integrity Check (MIC) functions
src/bcmcrypto/hmac.c	Keyed-Hashing for Message Authentication
src/bcmcrypto/aeskeywrap.c	Perform RFC3394 AES-based key wrap and unwrap functions

File	***Description***
src/bcmcrypto/md5.c*	MD5 source code
src/bcmcrypto/passhash.c*	WPA and IEEE 802.11i key hash algorithm
src/bcmcrypto/rijndael-alg-fst.c*	AES (Rijndael cipher)
src/bcmcrypto/prf.c	pseudo-random function used by WPA and IEEE 802.11i
src/bcmcrypto/sha1.c	secure hashing algorithm
src/bcmcrypto/rc4.c	RC4 stream cipher
src/bcmcrypto/wep.c	Wep functions

Naming

- All wireless (WL) device driver OS-specific (per-port) files and functions use the **wl_** prefix.
- All WL device driver OS-independent (common) files and functions use the **wlc_** prefix.
- OSL is an abbreviation for OS abstraction layer and is a set of macro definitions providing a set of OS-specific primitives for use by common code.
- WSEC is an abbreviation for security and refers to the support of all key formats and algorithms.

Porting

Porting the wireless device driver is primarily an effort to create two files, xx_osl.h and wl_xx.c, for the target operating environment.

Note: It should not be necessary to modify or even understand any wlc_xx source file to port the wireless device driver. Contact Broadcom before modifying any of the OS-independent source files.

1. Modify **include/typedefs.h** to define the set of common type definitions.
2. Create **include/xx_osl.h** where xx is the name of the OS (details below).
3. Optionally create **shared/xx_osl.c** to support xx_osl.h.
4. Modify **osl.h** to **#include xx_osl.h**.
5. Create **wl/sys/wl_xx.c**, primary driver port C file (details below).
6. Create **wl_sys/wl_xx.h** tunables for the WL port.
7. Modify **wl_dbg.h** to **#include wl_xx.h**.
8. Create a new subdirectory under src/wl/ and a Makefile to build the new port.
9. Start compiling following the “Porting Plan” on page 20.

Data Structures

Table 2: Data Structures

File	*Description*
wlc_info_t	The vast majority of wireless driver states is maintained in the wlc_info_t structure declared by wlc.h. Since this is a common code, it is all port-independent.
wl_info_t	Each driver port is responsible for defining and maintaining a wl_info_t structure for a port-specific state. This will be as simple or complex as your OS demands.
struct scb	The Station Control Block is a common data structure declared n wlc.h, which maintains a per-station (per-MAC-address) state. There is an array of SCBs defined within the wlc_info_t and a hash used by wlc_scblookup() to lookup and, optionally, allocate SCB for each remote station.
macstat_t d11cnt_t	Many event counters are defined by the hardware and IEEE 802.11 specification. Most of these are printed by wlc_dump().

Wireless Common Functions

The wireless common (WLC) code provides 98% of the wireless driver functionality.

- wlc_chipmatch() indicates if the provided PCI vendor and device ID match one of the supported chips.
- wlc_attach() allocates and returns a pointer to an initialized wlc_info_t structure. Most wlc_xx() functions take a wlc_info_t pointer as their first argument.
- wlc_detach() frees a previously allocated wlc_info_t.
- All callbacks from WLC to WL (as defined in wl_export.h) pass the opaque (void*) wireless handle as their first argument.
- wlc_up() is called to bring the driver up to the operational state.
- wlc_down() is called to take the driver down to the non-operational (post-attach) dormant state.
- wlc_reset() is called to reset the common-code software and hardware state.
- wlc_init() is called to reset and reinitialize the common-code software and hardware.
- wlc_isr() is the first-level common interrupt service routine. It determines if our dot11 core is interrupting, saves the event state, clears the hardware interrupt signal, and returns an indication of whether it was our function that interrupted and if further processing is required.
- wlc_dpc() is the second-level common interrupt handler and should be called when wlc_isr() returns with wantdpc true.
- wlc_send() is the common transmit function.
- wlc_ioctl() supports over 100 IOCTL commands defined in include/wlioctl.h with the aim of eliminating direct manipulation of WLC-> fields by port drivers. Calling wlc_ioctl() whenever possible instead of manipulating WLC-> fields directly ensures better forward compatibility as the common code is enhanced. In addition, directly modifying WLC-> fields may cause various, possibly subtle, side effects in the common code.
- wlc_statsupd() is called to update (refresh) WLC statistical counter fields prior to referencing their values.

- wlc_dump() prints the wlc_info_t fields. It is useful for debugging.
- wlc_set() is an integer set interface for common wlc_ioctl handlers.
- wlc_disassociate_client() is called for the client to disassociate from an AP.

Driver Operational States

The driver has a number of operational states. These states are transitioned as the driver is loaded, brought into normal operation, and taken offline. Table 3 describes the operational states.

Timers are typically allocated/deallocated in attach/detach and enabled/disabled in up/down.

Table 3: Driver Operation States

State	***Description***
Unloaded	The lowest state is unloaded. The driver is not loaded, and no resources are allocated.
Attached	Loading the driver and attaching each recognized device results in the down state. Resources are allocated, but the device is not running until it is powered on. Actions are best performed once driver load time is located in attach routines. Routines include data structure allocation and initialization, DMA ring allocation, shared mappings established, interrupt handlers registered, and so on.
Detach	Detach is anti-attach. It asserts down, then deallocates resources allocated by attach.
Down	Down is anti-up. Everything that up did, down undoes. Down starts with a reset, then clears all software states created by up including associations, timers, and other SW state machines. Down should result in SW and HW states identical to the original down after load + attach.
Up	Marking the device up expresses an intent that the driver should be operational. While up, the driver may need to recover from various runtime errors, some of which require reset and reinitialization of some portions of the software and hardware state. While the device is up, it can temporarily be non-operational if reset and reinit, but these are transient states.
Reset	Reset and initialization occur exclusively in the up state. Reset puts the device into reset state, takes it out of reset, performs a very few number of initializations to support some minimal features we need when the device is down (MDIO access, LEDs on), and reclaims any SW resources (DMA descriptors and I/O buffers) no longer posted to the chip for DMA.
Init	Init always starts by calling reset, but reset may also be called alone (when down is called). Init always starts with a reset to put the device into a known state, then performs all the various initialization SW and HW steps to make the interface fully functional.

Microcode load is currently part of init although it may move in the future to wlc_up().

As part of power-savings, the WL driver turns off most of the chip when down. Specifically, the radio and analog core are disabled, d11 core is reset, and, for the 4306, the external oscillator and PLL are turned off. Only a portion of the PCI core is left powered on. The driver internal flag wlc->clk is true when the chip clocks are enabled and d11 core(s) are out-of-reset.

Note: For programming purposes wlc->up implies wlc->clk and !wlc->clk implies !wlc->up.

While the driver is down (!wlc->up), the clocks may be turned off and on using the **wl clk off/on** command . Complicating this, each state transition action is typically partitioned into a per-port routine (WL) and one or more common routines (wlc). The per-port routine does very little before/after calling the common routine. The common routine is almost always further partitioned into SW-state centric functions and HW-state-chip (actually I/O-core) centric functions.

State Model:

The state model followed by the WL driver is:

- The hardware is just a temporary cached copy of the software state.
- The WL driver always writes through the software state into the hardware.

All operations that affect the state model, modify the software state and, optionally, are applied to the hardware. The hardware might not be available at the time the state is modified (driver is down, clocks are off, and so on).

When it becomes available in the future, the driver applies the current software state to it. The hardware cached state comes and goes—WL may reset the core, reinitialize individual blocks within the core, turn off PLLs and oscillators— but the software state persists. Only a driver unload clears all of the software state.

OS Abstraction Layer

The following primitives must be defined in xx_osl.h.

Table 4: OS Abstraction Layer

Function	***Description***
ASSERT(exp)	When expression evaluates to TRUE, an error condition has occurred. A debug driver will halt operation while citing the error. Refer to the VxWorks® or the Linux example port for implementation examples.

OSL_PCMCIA_READ_ATTR()
OSL_PCMCIA_WRITE_ADDR()
	Required if PCMCIA mode is supported.

OSL_PCI_READ_CONFIG()
OSL_PCI_WRITE_CONFIG()
	Required if PCI mode is supported.

printf()	This is the kernel equivalent to printf (3C).

sprintf()
strcmp()
strncmp()
strlen()
strcpy()
strncpy()
	Kernel equivalents to these stdio operations.

R_REG()
W_REG()
	Used to read and write device registers.

AND_REG()
OR_REG()
	Defined in terms of R_REG() and W_REG().

Table 4: OS Abstraction Layer (Cont.)

Function	Description

bcopy()
bcmp()
bzero()
	Kernel equivalent to these libc routines.

MALLOC()
FREE()
	Called by the driver to allocate software structures.
	This is not used to allocate DMA-able memory.

OSL_UNCACHED()	Convert kernel-virtual address into uncached virtual address.

REG_MAP()
REG_UNMAP()
	Map device registers and return a kernel-virtual address.

BUS_SWAP32()	Host/bus architecture-specific 32-bit swap.
BUSPROBE()	Safe way to issue a device read that may target-abort.

DMA_ALLOC_CONSISTENT()
DMA_FREE_CONSISTENT()
	Called by the driver to allocate DMA descriptor rings. This memory should be DMA-able and should not require explicit software processor cache flushing.

DMA_TX DMA_RX	Map/unmap direction.

DMA_MAP()
DMA_UNMAP()
	Maps a packet buffer prior to DMA. DMA_MAP() takes a kernel-virtual address and length and a flag indicating the direction of data (PCI_DMA_TODEVICE or PCI_DMA_FROMDEVICE), performs any necessary pre-DMA cache flushing , and returns an IO address suitable for direct DMA use by the device. DMA_UNMAP() performs any necessary post-DMA cache flushing and frees any resources allocated by DMA_MAP().

OSL_DELAY()	Spin wait for N microseconds.

R_SM()
W_SM()
BZERO_SM()
	Reads 32 bits, writes 32 bits, and bzero N bytes of shared DMA-able memory allocated from DMA_ALLOC_CONSISTENT() or DMA_MAP().

PKTGET()	Allocates a packet buffer. Multiple non-contiguous packet buffers may be chained to represent a single packet. Packets are treated by driver common code as void *p. The native OS-specific packet structure or a driver-specific wireless packet structure may be used.
PKTFREE()	Free a previously allocated void *p packet buffer.
PKTDATA()	Returns the kernel virtual address of the first byte of the packet buffer.
PKTLEN()	Returns the number of bytes of data starting at PKTDATA() contained in the packet buffer.
PKTNEXT()	Returns the value of the next packet buffer in the chain or NULL.
PKTSETLEN()	Sets the length of data bytes in this packet buffer to N bytes.
PKTPUSH()	Subtracts N bytes from the packet buffer starting data address and adds N bytes to the packet buffer length. Returns the new starting data address.
PKTPULL()	Adds N bytes to the packet buffer starting data address and subtracts N bytes from the packet buffer length. Returns the new starting data address.

Table 4: OS Abstraction Layer (Cont.)

Function	*Description*
PKTDUP()	Creates a logical copy of a packet buffer. This may copy data or only increment a data buffer reference count.
PKTCOOKIE()	Returns the opaque void* cookie previously associated with the packet buffer using PKTSETCOOKIE() or NULL.
PKTSETCOOKIE()	Binds an arbitrary void* cookie to the packet buffer.
PKTLINK()	Returns the value of the packet buffer link field, which points to the next packet (chain) in a queue of packets. Used by the bcmutils.h pktenq()/pktdeq() functions.
PKTSETLINK()	Sets the packet buffer link field to point to the next packet (chain) in a queue of packets.
osl_init()	Sets OSL initialization.
OSL_GETCYCLES(x)	Gets the processor cycle count.

OS Abstraction Layer Packet Buffers

The OSL PKT* routines provide a generic set of primitives for operating on packet buffers. The common code achieves OS independence by treating these packet buffers as void *p and using the PKT* primitives exclusively for procedural access to their content.

The choice of underlying PKT representation is left up to each driver port. Defining the PKT primitives directly in terms of the underlying native OS packet is usually the simplest approach. The driver architecture also allows for the PKT implementation and underlying native OS packet design to differ.

PKT buffers are objects containing zero or more bytes of data, may be chained (PKTNEXT) into multiple noncontiguous buffers representing a single packet, and queued (PKTLINK) onto queues of packet chains. PKT buffers can be allocated and freed. Data may be prepended to an existing PKT buffer using PKTPUSH(). Leading data may be removed from an existing PKT buffer using PKTPULL(). Trailing data may be removed from an existing PKT buffer using PKTSETLEN().

Wireless Packet Buffer Naming Conventions

The wireless device driver accepts Ethernet or IEEE 802.3/SNAP-encapsulated packets (MSDUs) from the TCP/IP stack converting each into one or more frames (Fragments or MPDUs) for transmission. One or more received fragments are reassembled into an Ethernet packet before being sent up to the TCP/IP stack or forwarded to another station if the host is operating as an AP.

The wireless common code manipulates two types of packet buffers. The code void *msdu are opaque handles for the native OS type. These are treated as foreign objects by the WLC code and are NOT subject to access by the PKT* primitives. The codes, void *mpdu and void *p, are OSL PKT type handles and can be manipulated via the PKT* primitives.

GMODE

This section describes the GMODE values used by the driver's WLC_SET_GMODE ioctl handler, and explains the design decisions that went into the setups and the implication of those choices.

```
/* 54g modes (basic bits may still be overridden) */
#define GMODE_LEGACY_B       0   /* Rateset: 1b, 2b, 5.5, 11 */
                                 /* Preamble: Long */
                                 /* Shortslot: Off */
#define GMODE_AUTO           1   /* Rateset: 1b, 2b, 5.5b, 11b, 18, 24, 36, 54 */
                                 /* Extended Rateset: 6, 9, 12, 48 */
                                 /* Preamble: Long */
                                 /* Shortslot: Auto */
#define GMODE_ONLY           2   /* Rateset: 1b, 2b, 5.5b, 11b, 18, 24b, 36, 54 */
                                 /* Extended Rateset: 6b, 9, 12b, 48 */
                                 /* Preamble: Short required */
                                 /* Shortslot: Auto */
#define GMODE_B_DEFERRED     3   /* Rateset: 1b, 2b, 5.5b, 11b, 18, 24, 36, 54 */
                                 /* Extended Rateset: 6, 9, 12, 48 */
                                 /* Preamble: Long */
                                 /* Shortslot: On */
#define GMODE_PERFORMANCE    4   /* Rateset: 1b, 2b, 5.5b, 6b, 9, 11b, 12b, 18, 24b, 36, 48, 54 */
                                 /* Preamble: Short required */
                                 /* Shortslot: On and required */
#define GMODE_LRS            5   /* Rateset: 1b, 2b, 5.5b, 11b */
                                 /* Extended Rateset: 6, 9, 12, 18, 24, 36, 48, 54 */
                                 /* Preamble: Long */
                                 /* Shortslot: Auto */
#define GMODE_MAX            6
```

Broadcom’s default rateset for 11g is {1b, 2b, 5.5b, 11b, 6, 9, 12, 18, 24, 36, 48, 54}, where b means Basic. These are all the 11b and 11g rates, with all 11b rates marked as Basic. The significance of all 11b rates being Basic is that since all STAs joining a network must support the Basic rates, *old* IEEE 802.11 (non-11b) devices that only support 1 Mbps and 2 Mbps cannot join. By including 5.5 Mbps and 11 Mbps as Basic, there is a 3% TCP performance improvement at 11 Mbps compared to 1 Mbps and 2 Mbps as the only basic rates.

The rates are typically split between the original IEEE 802.11 supported rates IE (Information Element), ID 3, and the new IEEE 802.11g extended supported rates, ID 50, like this:

Supported Rates: 1b, 2b, 5.5b, 11b, 18, 24, 36, 54

Ext. Sup. Rates: 6, 9, 12, 48

The choice of split is important because existing IEEE 802.11b devices cannot access the Extended Supported Rates (ESR) element. Also, Broadcom’s early 54g™ drivers did not access the ESR element. A straight forward split would put 1, 2, 5.5, 11, 6, 9, 12, and 18 in the first element and 24, 36, 48, 54 in the ESR. For Broadcom’s 54g™ drivers, that would limit their maximum rate to 18 since they do not access the ESR. The available split allows a legacy 54g™ driver to see OFDM rates 18, 24, 36, and 54, so there is a good range of rates from which to select, with the maximum rate is 54. Also, some IEEE 802.11b drivers may look no further than the first 4 elements of the supported rates, so all CCK rates should appear first.

For the CCK preamble setting, many existing IEEE 802.11b devices do not support Short Preamble. If short preamble is set for a network, it allows association only by short preamble capable STAs. So typically Broadcom's 11b/11g mixed network modes do not enforce the short preamble requirement. Since 11g APs include the ERP IE, the Barker Preamble Mode bit allows short preamble operation until a long preamble only capable STA joins, even if the network is not advertising short preambles in the Beacon's capability field.

Only CCK rates are allowed in the network, and only 1 Mbps and 2 Mbps are basic so legacy IEEE 802.11 devices can join. In this mode, the 11g AP or IBSS will not include an ERP IE or an ESR IE. This mode is supposed to look as much like an early IEEE 802.11b network to allow interoperability with devices that have trouble with any of the newer specification changes.

All 11g rates are available, but only CCK rates are basic to allow IEEE 802.11 devices to join. The rateset is split with only 4 rates in the supported rates IE to allow interoperability with IEEE 802.11 devices that have trouble with more than 4 rates in the IE.

ERP and ESR IEs are present, so the Barker Preamble Mode bit allows short preamble operation when possible. This is true of all the following GMODEs:

GMODE_LEGACY_B:

Rateset: 1b, 2b, 5.5, 11

Preamble: Long

Shortslot: Off

GMODE_LRS:

Rateset: 1b, 2b, 5.5b, 11b

Extended Rateset: 6, 9, 12, 18, 24, 36, 48, 54

Preamble: Long

Shortslot: Auto

GMODE_AUTO:

Rateset: 1b, 2b, 5.5b, 11b, 18, 24, 36, 54

Extended Rateset: 6b, 9, 12b, 48

Preamble: Auto

Shortslot: Auto

NMODE

For Mimo/11n chipsets, there is a new control parameter NMODE in the driver. If set, IEEE 802.11n features will be enabled. It can only be set as TRUE when running on N-Phy hardware.

LEDs

Most Broadcom wireless cards support LEDs connected to the first 4 GPIO pins (GPIO 0–3). MiniPCI cards generally route:

- GPIO #0 to MPCI #11
- GPIO #1 to MPCI #12
- GPIO #2 to MPCI #14
- GPIO #3 to MPCI Active_L

Refer to the particular card schematics for details.

wlc_led.c supports associating each of these GPIO pins with a behavior from the set defined in wlioctl.h:

Table 5: LED Functions

LED Function	***Behavior***	***Description***
WL_LED_OFF	0	Always off
WL_LED_ON	1	Always on, if driver is loaded.
WL_LED_ACTIVITY	2	• Medium blink when there is activity (data frames)on the LAN. An STA must be associated to exchange data frames. An AP must have at least one associated STA to exchange data frames. • Turn the LED OFF if the driver is down for a reason other than MPC(power saving).
WL_LED_RADIO	3	ON if either a 2.4 GHz or 5 GHz radio is enabled.
WL_LED_ARADIO	4	• ON if the 5 GHz PHY is present, the driver is up (active), and the driver is configured in either IEEE 802.11a or dual-band mode. • Turn the LED OFF if the driver is down for a reason other than MPC (power saving).
WL_LED_BRADIO	5	• ON if the 2.4 GHz (IEEE 802.11b or 802.11g) PHY is present, the driver is up (active), and the driver is configured in either IEEE 802.11b/g or dual-band mode. • Turn the LED OFF if the driver is down for a reason other than MPC (power saving).
WL_LED_BGMODE	6	• STA: ON when associated to an AP with OFDM rates set on the 2.4 GHz radio. • Turn the LED OFF if the driver is down for a reason other than MPC (power saving). • AP: ON when including OFDM rates in the rate set (IEEE 802.11g mode).
WL_LED_WI1	7	ON if either a 2.4 GHz or 5 GHz radio is enabled. Medium blink when there is activity.

Table 5: LED Functions

LED Function	*Behavior*	*Description*
WL_LED_WI2	8	• STA: ON when associated with an AP. • STA: OFF if the radio is disabled. • STA: Slow blink if not associated with an AP. • STA: Fast blink if there is data activity. • AP: ON by default. Fast blink if there is activity.
WL_LED_WI3	9	• STA: ON when associated with an IEEE 802.11b AP having CCK-only rates in the rate set. • STA: OFF if the radio is disabled. • STA: Slow blink if not associated to an AP. • STA: Fast blink if there is data activity. • AP: ON when the AP is advertising IEEE 802.11b CCK-only rates.
WL_LED_ASSOC	10	STA: Slow blink when unassociated (even MPC driven down). ON, if associated with an AP.
WL_LED_NULL/ WL_LED_INACTIVE	11	Null behavior: clears the current state.
WL_LED_ASSOCACT	12	ON when associated (OFF when not associated); blink medium for activity.
WL_LED_WI4	13	• When the driver is down, turn the LED OFF when associated. • If the driver is up, and the 5 GHz radio is enabled, associated, and there is activity, blink fast. • If no activity, turn ON, but no blinking. • If associated, but the 5 GHz radio is not on, then turn the LED and blinking OFF. • If not yet connected to the AP, blink slow.
WL_LED_WI5	14	• When driver is down, turn OFF the LED. • If driver is up, and the 2.5 GHz radio is enabled, associated, and there is activity, blink fast. • If no activity, turn the LED OFF and do not blink. • If associated, but the 2.4 GHz radio is not on, turn the LED ON and blinking OFF. • If not yet connected to the AP, blink slow.
WL_LED_BLINKSLOW	15	• Slow blink (500 ms ON, followed by 500 ms OFF)
WL_LED_BLINKMED	16	• Medium blink (80 ms ON, followed by 20 ms OFF)
WL_LED_BLINKFAST	17	• Fast blink (40 ms ON, followed by 40 ms OFF)
WL_LED_BLINKCUSTOM	18	• Custom blink + + (200 ms ON, followed by 100 ms OFF)
WL_LED_ASSOC_WITH_SEC	20	• When connected with security enabled, keep ON for 300 seconds
WL_LED_START_OFF	21	• OFF during boot: can be turned on later.

Override either by NVRAM variables or (Windows) registry entries.

NVRAM format:

ledbhX = Bits[Bit7=polarity, Bit6:0=behavior] where: X = LED/GPIO number Polarity: 0 = active high; 1 = active low

(SPROM words 50 and 51 are reserved for WL LED behavior).

Windows registry entries format:

```
gpio%d[_l]=%d
```

WL_XX.C

Choose an existing WL port to base the new port upon. The current supported ports are:

- wl_linux.c
- wl_vx.c

Create a new wl_xx.c that defines the minimal driver OS-specific entry points, which call into wlc_xx() routines. This file must also define the set of WLC-to-WL callbacks listed in wl_export.h.

The program, wl_xx.c, is responsible for defining and maintaining any port-specific per-instance state— wl_info_t— and referencing.

Compile Time Flags

- IL_BIGENDIAN: always define for big-endian platform.
- BCMDBG: enable debugging code (ASSERT, msglevel, dump, and so on).
- AP: enable support for wireless AP functionality.
- STA: enable support for wireless STA and IBSS functionality.
- APSTA: enable both AP and STA wireless functionality.

Debugging

- Compile with BCMDBG.(add DEBUG=1 in wl_default located in src/wl/config)
- Familiarize yourself with the wl command.
- Use wl msglevel xx to enable various levels of diagnostic printing (see wl_dbg.h).
- Use wl dump to print wl_info_t and wlc_info_t state.
- Use wl scbdump to print scb state.

Porting Plan

1. Implement the OS abstraction layer and port-specific routines defined in wl_export.h.
2. Without hooking up receive or transmit paths to stack, monitor then receives packets and sees beacons and probe responses. This verifies chip communications and exercises most of the OSL.
3. Hook up configuration support I/O controls and join or find an IBSS.

   Joining or finding an IBSS verifies the I/O controls (IOCTL) path in general. Also, once part of an IBSS, beacons sent by Broadcom card can be seen. Successful beacon transmission validates the chip template transmit path.
4. Debug print receive data packets from another station.

   The data packets can now be received and the Address Resolution Protocol (ARP) requests can be tested. The ARP requests broadcast MAC address or PINGs to unicast MAC address by using static ARP entries on another machine.
5. Test transmit control path.

   The driver sends probe responses from Broadcom card when set up as an IBSS. By seeing the probe responses, the receive to transmit loop at the low end of the driver is verified. It can also be tested by associating with an AP to verify the receive/transmit control exchanges work for authentication and association.
6. Hook up the receive path and observe ARP requests or other packets going to the stack. The stack transmit packets can also be seen from the stack for replies.
7. Hook up transmit path and close the loop on ARP or PING.
8. Verify security capability (WEP, TKIP, AES).

Connecting everything®

BROADCOM®

BROADCOM CORPORATION
5300 California Avenue
Irvine, CA 92617

Phone: 949-926-5000
Fax: 949-926-5203
E-mail: info@broadcom.com
Web: www.broadcom.com

43XX-AN1300-R May 13, 2011